ISO TC184/SC4/WG11/N002

Date: August 21, 1996

Supercedes SC4/

PRODUCT DATA REPRESENTATION AND EXCHANGE

Part: ____ **Title:** EXPRESS-X Reference Manual

Purpose of this document as it relates to the target document is:

- X **Primary Content**
- ___ **Issue Discussion**
- ___ **Alternate Proposal**
- ___ **Partial Content**

Current Status: Working

ABSTRACT: The intended use of EXPRESS-X is to define mappings between pairs of EXPRESS schemas, where one EXPRESS scheme represents an abstract view of the other. These mappings are defined in a declarative fashion. For example, EXPRESS-X can be used to implement the mapping of entities from the AIM of an Application Protocol to its ARM. The EXPRESS-X language controls these mapping by specifying the conditions under which a new view entity should be created, and how the attributes should be derived for that new view entity. EXPRESS-X combines the EXPRESS-V language (ISO TC184/SC4/WG5 N251) with the EXPRESS-M language (ISO TC184/SC4/WG5 N243).

KEYWORDS:
Mapping Language
View
Schema
EXPRESS
Database

Document Status/Dates

Part Documents		Other SC4 Documents	
X	**Working Draft**	8/21/96	**Working**
	Project Draft		**Released**
	Released Draft		**Confirmed**
	Technically Complete		**Approved**
	Editorially Complete		
	ISO Committee Draft		

Owner/Editor:
Address: Lab for Industrial Information Infrastructure
Rensselaer Polytechnic Institute
CII Building, Room 7015
Troy, New York 12180-3590
USA

Telephone/FAX: +1 (518) 276-6751 / +1 (518) 276-2702
E-Mail: rose@rdrc.rpi.edu

Alternate:
Address:

Telephone/FAX:
E-Mail:

Comments to Reader

This version of the specification is compiled from the contributions of several people. There is ample room for improvement not only in individual sections but also in ensuring the consistency between sections. This version remains technically incomplete.

Contributors

Ji Wen	Rensselaer Polytechnic Institute
Martin Hardwick	Rensselaer Polytechnic Institute
David L. Spooner	Rensselaer Polytechnic Institute
Craig Schlenoff	Rensselaer Polytechnic Institute
John Valois	STEP Tools, Inc.
Ian Bailey	CIMIO Ltd.

Meetings

Table of Contents

Foreword

This document describes the EXPRESS-X language, which currently is not an official Part of ISO 10303. The document has been prepared by the Laboratory for Industrial Information Infrastructure at Rensselaer Polytechnic Institute, who developed the EXPRESS-V language (ISO TC184/SC4/WG5 N251). It incorporates concepts from the EXPRESS-M language developed by CIMIO, Ltd. (ISO TC184/SC4/WG5 N243).

This is a Working Draft.

The EXPRESS-X language described in this document is related to a series of Parts which together comprise the International Standard ISO 10303 Industrial Automation Systems - Product Data Representation and Exchange. The Parts are as follows:

-- ISO 10303-1 Overview and Fundamental Principles;

-- ISO 10303-11 Description Methods: The EXPRESS Language Reference Manual;

-- ISO 10303-21 Clear Text Encoding of the Exchange Structure;

-- ISO 10303-22 STEP Data Access Interface Specification

-- ISO 10303-31 Conformance Testing Methodology & Framework: General Concepts;

-- ISO 10303-41 Integrated Generic Resources: Fundamentals of Product Description and Support;

-- ISO 10303-42 Integrated Generic Resources: Geometric and Topological Representation;

-- ISO 10303-43 Integrated Generic Resources: Representation Structures;

-- ISO 10303-44 Integrated Generic Resources: Product Structure Configuration;

-- ISO 10303-46 Integrated Generic Resources: Visual Presentation;

-- ISO 10303-101 Integrated Application Resources: Draughting;

-- ISO 10303-201 Application Protocol: Explicit Draughting;

-- ISO 10303-203 Application Protocol: Configuration Controlled Design.

The reader may obtain information on these Parts of ISO 10303 from the ISO Central Secretariat.

1 Introduction to EXPRESS-X

ISO 10303 is a series of International Standards for the computer-sensible representation and exchange of product data. The objective is to provide a mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system. The nature of this description makes it suitable not only for file exchange, but also as a basis for implementing and sharing product databases and archiving.

Each International Standard in the ISO 10303 series is published as a separate Part. Parts are grouped into one of the following classes: description methods, integrated resources, application protocols, implementation forms, and conformance testing. The classes are described in ISO 10303-Part 1.

This document describes the EXPRESS-X language, which can be used to define mappings between entities from one EXPRESS schema to entities in another schema that represents an abstract view of the first. This satisfies an industrial need to easily tailor information models to meet the needs of individual application systems.

Major subdivisions in this reference manual are:

-- Introduction to EXPRESS-X

-- Fundamental Principles

-- Language Definition

-- Examples of EXPRESS-X

-- Syntax Rules for EXPRESS-X

The remainder of this introduction provides the reader with background on the EXPRESS-X concept and the definitions of key terms.

1.1 Motivation for EXPRESS-X

By its nature, a representation and exchange standard for product data such as STEP must be complete and unambiguous. As a result, it is large and contains details that many individual application systems will not need. In other words, it is the union of the requirements of these application systems. This implies that a simplified view of a product model that omits unnecessary details of the model should be sufficient for many applications. Using such a simplified view is desirable for these application systems, since such a view is conceptually easier to understand and process within the application system. This is especially true for legacy systems.

Unfortunately, the optimal simplified view of a product model for one application system may not be the optimal view for another application system, even if the two systems are related. As a result, there is a need to be able to easily create views of product models that are tailored to individual application systems.This will in general improve the usability of the STEP standards in many situations.

In STEP, a product model is defined using EXPRESS. This means that a view of a product model must be based on the EXPRESS definition of that product model. Thus, for STEP, a language is needed that facilitates definitions of views of EXPRESS information models. This is the purpose of EXPRESS-X. It is an extension of EXPRESS that includes constructs for defining views of EXPRESS information models.

Thus, the goal of the EXPRESS-X language is to define mappings between information models defined in EXPRESS as shown in Figure 1. An implementation of the EXPRESS-X language must include a compiler for validating the syntax of an EXPRESS-X definition and a run-time system for materializing a view.

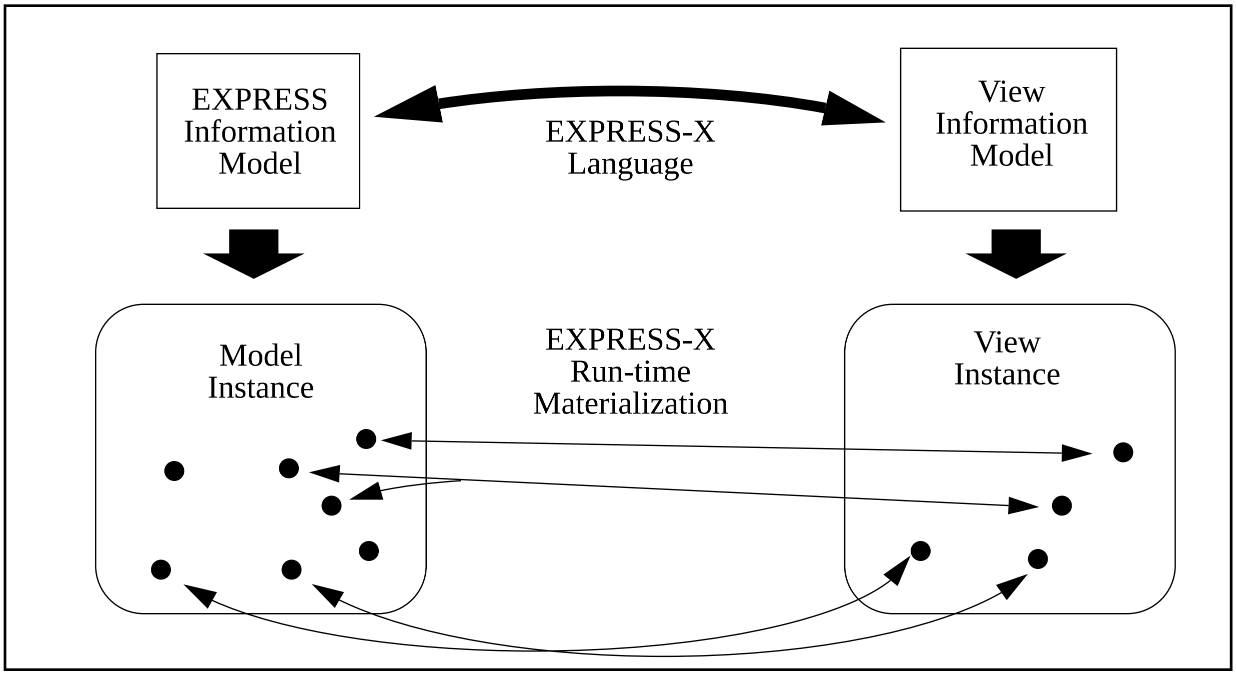

FIGURE 1. **EXPRESS-X Overview**

1.2 What is EXPRESS-X

EXPRESS-X allows one to create alternate representations of EXPRESS models and mappings between EXPRESS models and other applications (e.g., IGES). These alternate representations are called *views* of the original models. The algorithm for deriving the entity types in a view from the entities in an original EXPRESS model is specified using various types of mapping declarations.

Creation of a view of an EXPRESS model requires two phases: materialize and compose. In the materialize phase, the view entity instances are created, along with those attributes of the new view instances that depend only on data from the entities in the original EXPRESS model. In the compose phase, attributes of the new view instances that depend on other view instances, and hence could not be initialized during the materialize phase, are created. An example of such an attribute is one that represents a relationship between view instances. More than one pass may be needed in the compose phase if complex dependencies exist between the attributes.

1.3 Updating Views -- Two-Way Mappings

In many situations, it is desirable to allow changes made to the entity instances in a view to be mapped back to the original EXPRESS model from which the view was created. This can be done in EXPRESS-X by defining a second set of mappings (i.e., a second **`SCHEMA-MAP`** as defined in Chapter 3) that maps from a view back to the original model. In this case, since the entity instances already exist in the original model, only a compose phase is needed.

An example of updating views is given in Appendix E.

1.4 Definitions

The EXPRESS-X language uses terminology consistent with that of EXPRESS whenever possible. Definitions of terms that are not part of EXPRESS follow:

View: An abstraction of an information model tailored for some application system or user that omits unnecessary details and reorganizes the remaining information into a more easily used form for the application or user.

Base Schema: An EXPRESS information model.

Base Model: An instantiation of a base schema.

Base Entity Type: An entity type defined in a base schema.

Base Instance: An instance of an entity type defined in a base schema.

View Schema: An EXPRESS information model that defines entities derived from the entities in a base schema.

View Model: An instantiation of a view schema.

View Entity Type: An entity type defined in a view schema.

View instance: An instance of an entity type defined in a view schema.

Materialize: The process of creating a view model from a base model.

Mapping Schema: An EXPRESS-X schema that defines the detailed algorithms for mapping the entity types from a base schema to a view schema.

Mapping: A declaration in a mapping schema that defines the algorithm for mapping a base entity type to a view entity type.

2 Fundamental Principles

In database terminology, a *view* is a perspective of a database. There may be many views for a given physical database, each view tailored to the requirements of a particular application program or user. A view may omit parts of the database that are of no interest to the application system or user for which the view was created. It may reorganize the database by changing its structure and/or the data types of the data it contains. The goal of creating a view is to simplify the use of the database by the application system or user for which the view was created.

2.1 Logical Organization for an EXPRESS-X Specification

The specification of a view using EXPRESS-X requires the definition of three schemas, two of which are ordinary EXPRESS schemas (see Figure 2). The first of these is called the *base schema* and defines the schema for the original product model from which the view will be derived. The second of these is the *view schema* which defines the product model for the materialized view - i.e., the entity types that will be in the view and the attributes for each of these entity types. Both these schemas are defined as ordinary EXPRESS schemas.

The third schema is the mapping schema and is defined using the EXPRESS-X language. The mapping schema defines mappings between entities in the base schema and the view schema. Each mapping specifies some or all of the following information:

- A group of entity types in the base schema from which an entity type in the view schema is created,
- A predicate defined over this group of entity types from the base schema that specifies the conditions that must be true for a new instance of the view entity type to be created, and
- Specifications of how the values for each of the attributes of a new view instance are to be computed once the new view instance is created.

2.2 The Mapping Schema

As shown in Figure 2, a mapping schema defines a relationship between information models defined as **`SCHEMA`**’s in EXPRESS. The mapping schema itself is also an information model defined as a **`SCHEMA_MAP`** in EXPRESS-X. A mapping schema (i.e., **`SCHEMA_MAP`**) is defined so that:

- one can better understand the relationship between the two schemas, and
- an information processing system (e.g., an EXPRESS-X compiler) can create a data processing system that will convert information belonging to one of the schemas (i.e., the base schema) into information belonging to the other schema (i.e., the view schema).

The first role is considered the more important role. A good EXPRESS-X mapping schema defines the relationship between two schemas in a way that is simple and easy to understand. If a particular mapping cannot be described in a straightforward manner in EXPRESS-X, then it may be represented as a mapping with only a comment in its declaration that describes informally the mapping that would be defined if sufficient resources existed to produce it.

For example, if a mapping declaration requires a statistical analysis that can only be performed using advanced numerical techniques, then the body of the declaration may contain only a comment that provides a reference to the algorithm in the literature.

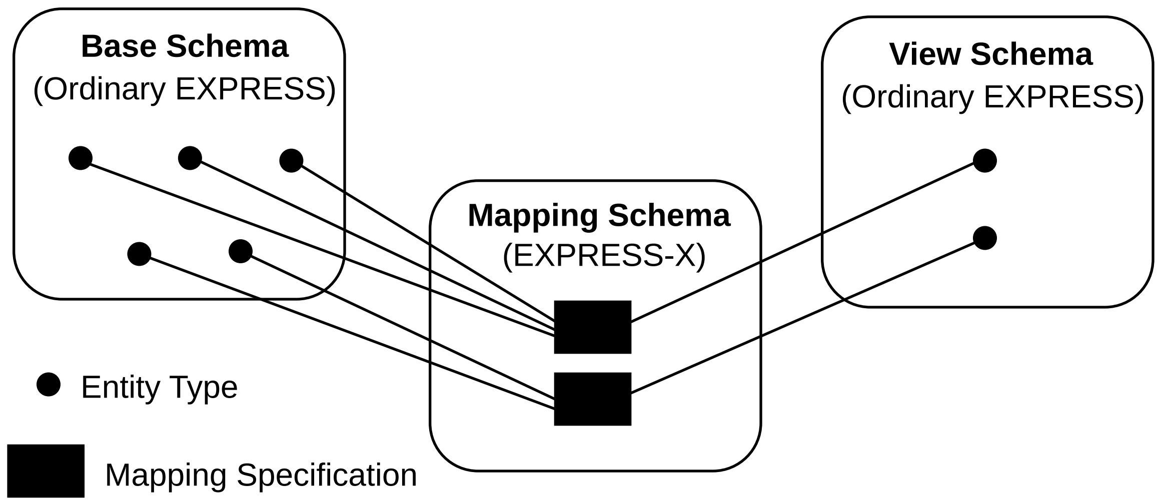

FIGURE 2. **Three Schemas in an EXPRESS-X Specification**

Conceptually, it is convenient to think of a mapping schema as defining mappings (see section 2.4 below) between a base schema and a view schema. In practise, however, a mapping schema can define mappings between any set of entity types, independent of the schema or schemas from which they come. In fact, a mapping schema can reference entity types from many schemas, not just two. It is also the case that the EXPRESS-X language has no construct to specify which schema is a base schema and which is a view schema.

The concepts of base schema and view schema will be used throughout the rest of this manual to simplify the explanation of how the various constructs in the EXPRESS-X language work. In most examples in the manual two schemas are used, one that plays the role of a base schema and one that plays the role of a view schema. However, it is important to keep in mind that there are no physical restrictions on any of the constructs in the EXPRESS-X language with respect to the schemas on which they operate.

2.3 Materializing a View

As illustrated in Figure 2, an EXPRESS-X specification defines a mapping between a group of entity types in the base schema and a group of entity types in the view schema. To materialize a view model (i.e., an instantiation of a view schema) from a base model (i.e., an instantiation of a base schema), each of the mapping specifications must be applied to the appropriate entity instances in the base model. This requires applying the mapping to all combinations of base instances that participate in that mapping.

For example, the top mapping in Figure 2 is defined between three entity types in the base schema and one in the view schema. To materialize this mapping, it is necessary to consider every combination of an entity of the first type in the base schema with an entity of the second and third types in the base schema. For each combination of entities, the predicate in the mapping that defines the conditions for creation of a view instance must be evaluated. If the predicate evaluates to true, a new view instance is created.

Once a new view instance is created in the view model, it is necessary to compute values for its attributes. In many cases, these values can be derived directly from the attribute values of the base instances from which the view instance is created. In these cases, a simple assignment statement that defines the derivation computation is all that must be specified for each attribute. In other cases, however, the computation of a value for an attribute may not be as straightforward. Consider, for example, two view instances that are related in some way, and this relationship is modeled by having one or both of the view instances point to the other. When the first view instance is created, the second may not yet exist in the view model. If the first view instance is to point to the second, then the attribute of the first view instance that is to point to the second must remain temporarily uninitialized. Its value must be initialized at a later time once the second view instance has been created.

2.4 Specification of Mappings

A mapping schema in EXPRESS-X (i.e., a **`SCHEMA_MAP`**) defines the relationship between two information models using three types of declarations: **`VIEW`**, **`COMPOSE`** and **`MEMBER`**. An information processing system uses these three types of declarations to create data processing functions that automate mappings between a set of schemas.

2.4.1 VIEW Declaration

A **`VIEW`** declaration specifies how to construct a particular entity type. It contains a **`FROM`** clause that identifies the base entity types from which the new entity type is created. It contains a **`WHEN`** clause that specifies the conditions that must be true for a new instance to be created. And it contains a body that specifies how to compute the values of the attributes for new instances.

Logically, the **`FROM`** clause creates an iteration over all combinations of instances for the entity types it lists. For each combination, the condition in the **`WHEN`** clause is evaluated. If true, a new instance is created.

For example, consider the following declaration:

```
VIEW v : vdb::ViewEntity
FROM (ba : bdb::BaseA, bb : bdb::BaseB)
WHEN (ba.attr1 > bb.attr2) AND (ba.attr2 > 0));
BEGIN_VIEW
         v.v_attr1 := ba.attr1;
         v.v_attr2 := bb.attr2;
END_VIEW;
```

This declaration says that view entities of type **`ViewEntity`** are to be created from base entities of type **`BaseA`** and **`BaseB`**. The identifiers **`vdb`** and **`bdb`** are used to specify the view schema and the base schema, respectively, and are defined elsewhere in the mapping schema. The variables **`v`**, **`ba`**, and **`bb`** are implicitly declared in this view declaration to represent instances of entity types **`ViewEntity`**, **`BaseA`** and **`BaseB`**, respectively. The **`FROM`** clause sets up an iteration over combinations of entity instances of type **`BaseA`** and **`BaseB`** in a base model. The **`WHEN`** clause creates a new view instance only for those combinations in which **`attr1`** of the **`BaseA`** instance is greater than **`attr2`** of the **`BaseB`** instance and in which **`attr2`** of the **`BaseA`** instance is greater than zero. The values for the attributes of a new view instance are copied from the attributes of the base instances.

2.4.2 COMPOSE Declaration

A **`COMPOSE`** declaration can be used in conjunction with a **`VIEW`** declaration when it is not possible to compute the values for all attributes of a view entity type when its instances are first created. As discussed above, it is sometimes necessary to perform multiple passes to compute the values for attributes when complex relationships exist between view entity types.

A **COMPOSE** declaration is much like a **VIEW** declaration except that it iterates over the instances of an existing entity type. Also, it does not create new instances as a **VIEW** declaration does; rather it computes values for attributes of existing instances of an entity type. The syntax is similar to a **VIEW** declaration except that the word “**COMPOSE**” replaces the word “**VIEW**” and the **FROM** clause is optional.

If a **COMPOSE** declaration contains a **FROM** clause, then the **FROM** clause creates an iteration over all combinations of the instances for the entity types that it lists along with all instances of the entity type being composed. If no **FROM** clause is used, then an iteration is created over just the instances of the entity type being composed. In either case, the **WHEN** clause restricts when the body of the **COMPOSE** declaration is applied.

2.4.3 MEMBER Declaration

A **MEMBER** declaration defines the entity types from a base schema that affect the value of an entity type in the view schema. In other words, a **MEMBER** declaration defines information about the relationships between two schemas (as do the other three types of declarations).

The **MEMBER** declaration has several uses in EXPRESS-X. For example, an information processing system may use a **MEMBER** declaration to specify when the value of a view entity type should be recomputed in response to changes in a base model. It may also use a **MEMBER** declaration to specify which entity types should be copied from a base model to a view model for a deep copy of an entity type that is mapped to a view.

A **MEMBER** declaration may also contain **FROM** and **WHEN** clauses. If present, they operate as they do for the **COMPOSE** declaration. That is, the **FROM** clause increases the combinations of instances to which the body of the **MEMBER** declaration is applied. The **WHEN** clause restricts these combinations to just those that satisfy the conditions imposed in the **WHEN** clause.

2.5 Conformance Levels

An EXPRESS-X mapping schema defines the relationships between a set of information models using a combination of **VIEW**, **COMPOSE**, and **MEMBER** declarations. A system using EXPRESS-X may choose to conform to the specifications using the following conformance classes.

Class 1

A Class 1 system processes only **VIEW** declarations with a **FROM** clause that contains a single base entity type. A system that conforms to this level must allow a user to apply a **VIEW** declaration to the instances of any single base entity type in a base schema. The result is the creation of view entity instances of a single view entity type belonging to a view schema.

Class 2

A Class 2 system processes **VIEW**, **COMPOSE** and **MEMBER** declarations. A system that conforms to this level must allow a user to apply **VIEW** and **MEMBER** declarations to one or more base entity instances. The result is the creation of one or more view entity instances.

Class 3+

Additional conformance classes are reserved for extensions to be defined during the ISO standard development process.

3 Language Specification Syntax

This section defines the syntax of the EXPRESS-X language using a notation that is similar to the Wirth Syntax used to define EXPRESS in ISO 10303 Part 11.

The base schema and the view schema are defined using standard EXPRESS and are not discussed further in detail. The mapping schema, which defines the mappings between the base and view schemas, is done using the new constructs in the EXPRESS-X language, and is discussed in detail in this section.

3.1 Basic Language Elements

The basic language elements for EXPRESS-X are similar to those in EXPRESS. An EXPRESS-X specification is composed of streams of text broken into physical lines composed of characters and ended by a newline character.

3.2 Character Set

See ISO 10303 Part 11, section 7.1 for details.

3.3 Keywords

The following EXPRESS-X keywords are not part of the EXPRESS language (they may be specified in upper, lower, or mixed case):

BEGIN_COMPOSE	**BEGIN_MEMBER**	**BEGIN_VIEW**	**COMPOSE**
DECLARE	**DELETE**	**END_COMPOSE**	**END_GLOBAL**
END_MEMBER	**END_SCHEMA_MAP**	**END_VIEW**	**EXCLUDE**
GLOBAL	**INSTANCE**	**IS**	**MEMBER**
NEW	**SCHEMA_MAP**	**VIEW**	**WHEN**

All other keywords in EXPRESS-X are defined as in EXPRESS (see ISO 10303 Part 11, section 7.2 for details).

3.4 Symbols

See ISO 10303 Part 11, section 7.3 for details.

3.5 The Logical Organization of an EXPRESS-X Specification

```
syntax = schema_decl { schema_decl } .
```

An EXPRESS-X specification consists of one or more mapping schemas, each of which defines the required view materialization process for a view of an EXPRESS model.

3.6 Defining Mapping Schemas

```
schema_decl = SCHEMA_MAP schema_id ';' schema_body END_SCHEMA_MAP ';' .
```

A mapping schema specification in EXPRESS-X is similar to a schema specification in EXPRESS.

```
schema_body = { interface_specification } [constant_decl] { global_decl }
```

```
                { declaration | rule_decl } .
```

The specification of the body of a mapping schema in EXPRESS-X has the same form as the specification of the body of a schema in EXPRESS, with two exceptions. When defining a mapping schema, it is necessary to create declarations that define the mappings. As a result, EXPRESS-X has an expanded set of allowable declarations for use in defining the mapping schema. It is also necessary in EXPRESS-X to include a global section that identifies the base schema and the view schema for the mappings.

3.7 Global Declarations in a Mapping Schema

The base and view schemas referenced in a mapping schema are declared in a global section at the beginning of the mapping schema. These declarations provide a unique name for each base schema and view schema used in the mapping schema. Among other things, these unique names are used throughout the mapping schema to qualify entity type names that are shared between multiple schemas.

```
global_decl = GLOBAL { schema_instance_decl | instantiation_clause }
              END_GLOBAL ';' .

schema_instance_decl = DECLARE schema_instance_id INSTANCE OF schema_id
                       ';' .

schema_instance_id = simple_id .

schema_id = simple_id .
```

The global section can also include instantiation definitions for instances of the entity types defined in the base and view schemas. Instances manually instantiated in this way are given names beginning with the character ‘#’ to distinguish them. The syntax for specifying a manually instantiated instance is taken from EXPRESS.

```
instantiation_clause = instance_id '=' entity_constructor ';'  .

instance_id = '#' extended_id  .

extended_id = [schema_id '::'] simple_id  .

entity_constructor = entity_ref '(' [ expression { ',' expression} ] ')'  .
```

The following is an example of a global declaration in a mapping schema.

```
GLOBAL

    (* schema instances *)
    DECLARE bdb INSTANCE OF base_schema;
    DECLARE vdb INSTANCE OF view_schema;

    (* manual instantiation
        The following creates two manual instances, 'hh' and 'ww' . These
        instances become part of the view model identified by 'vdb'.
    *)
    #vdb::hh = bdb::MALE('Tony Blurb', 39, #vdb::ww);
    #vdb::ww = bdb::FEMALE('Amanda DeCadanet', 25, #vdb::hh);
```

```
END_GLOBAL;
```

3.8 Other Declarations in a Mapping Schema

```
declaration = entity_decl | function_decl | procedure_decl | type_decl |
              view_decl | compose_decl | member_decl .
```

Other declarations in a mapping schema are similar to those in EXPRESS, with the addition of the three new types of declarations for defining mappings (i.e., **VIEW** declaration, **COMPOSE** declaration and **MEMBER** declaration).

Before going on, it is useful to see an example mapping schema. To show such an example, it is necessary first to define the base schema and view schema that will be used by the mapping schema. To do this, consider the following two schemas, one of which is named **Base_Schema** and the other **View_Schema**. Entities in the **View_Schema** schema will be derived from the entities in the **Base_Schema** schema when the view is materialized.

```
SCHEMA Base_Schema;

ENTITY BaseA;
    int_a1: INTEGER;
    str_a2: STRING;
END_ENTITY;

ENTITY BaseB;
    complex_b1: BaseA;
    str_b2    : STRING;
END_ENTITY;

ENTITY BaseC;
    real_c1: REAL;
    str_c2 : STRING;
END_ENTITY;

END_SCHEMA;
```

```
SCHEMA View_Schema;

ENTITY ViewEntity;
    int_v1 : INTEGER;
    real_v2: REAL;
    str_v3 : STRING;
END_ENTITY;

END_SCHEMA;
```

The skeleton of a mapping schema that defines the view materialization process for these two schemas is shown below. In this mapping schema, a **VIEW** declaration is used to specify how entities of type **ViewEntity** are created from the entity types in the base schema.

```
SCHEMA_MAP Mapping_Schema;

GLOBAL
      DECLARE bdb INSTANCE OF Base_Schema;
      DECLARE vdb INSTANCE OF View_Schema;
END_GLOBAL;

VIEW v : vdb::ViewEntity ;
. . .
END_VIEW;
```

```
END_SCHEMA_MAP;
```

The **VIEW** declaration in the mapping schema above defines the details of the mappings required to materialize a view of the base schema. The details for specifying these mappings are presented in the following sections.

3.9 The Logical Organization of a View Mapping Declaration

```
view_decl = view_head  [ algorithm_head ]  { stmt }  END_VIEW  ';'  .
```

A **VIEW** declaration specifies how base instances of one or more types are to be mapped to view instances. A **VIEW** declaration consists of a header and view statements. The purpose of the view header is to define the conditions under which a new view instance should be created in a view model from one or more base instances in a base model. The purpose of the view statements are to define how the values of the attributes for a newly created view instance are to be computed. The view declaration can also contain local definitions that will be needed by the view statements (i.e., **algorithm_head**).

```
view_head = VIEW general_head from_head when_clause BEGIN_VIEW .

general_head = ( (name_id FOR extended_entity_ref) | extended_entity_ref)
               ‘;’ .

name_id = simple_id .

extended_entity_ref = variable_id ':' parameter_type .
```

A view header begins with the keyword **VIEW** followed by the name of a view entity type defined in a view schema. This entity type name can be any valid extended entity reference (see below), and it defines the type of entity that is created in the view model by this view definition. Optionally, the entity type name can be preceded by a unique identifier and the keyword **FOR**. This is useful to uniquely identify **VIEW** declarations when more than one declaration is required for the same entity type.

An extended entity reference names an entity type defined in a schema. It begins with the declaration of a variable name to be used in the mapping declaration to refer to instances of the identified entity type. The variable name is followed by a colon (i.e., ‘`:`’) and a schema instance name defined in the global section of the mapping schema. The schema instance name is followed by an entity type name separated from the schema instance name with two colons (i.e., ‘`::`’). The entity type name must be declared in the schema identified by the schema instance name.

Examples of extended entity references are shown below:

```
b : bdb::BaseEntity ;

v : vdb::ViewEntity ;
```

The remainder of the view header contains a **FROM** clause and a **WHEN** clause. The former defines the base entity types in a base schema from which new view instances are to be materialized. The latter defines the conditions that must be true for the materialization of a view instance to be done. Both are discussed in detail in the next sections.

Inside the view mapping is a sequence of statements that defines how values for the attributes of a newly created view instance should be computed. These statements are described in a later section.

As an example, a complete mapping schema for the example started above is the following:

```
SCHEMA_MAP Mapping_Schema;

GLOBAL
         DECLARE bdb INSTANCE OF Base_Schema;
         DECLARE vdb INSTANCE OF View_Schema;
END_GLOBAL;

VIEW v : vdb::ViewEntity ;
FROM (ba : bdb::BaseA, bb : bdb::BaseB, bc : bdb::BaseC)
WHEN ((ba.int_a1 = bb.complex_b1.int_a1) AND
      (NOT (bc.real_c1 > 10.0)));
BEGIN_VIEW
   v.int_v1  := 100;
   v.real_v2 := -bc.real_c1;
   v.str_v3  := 'This is a view object';
END_VIEW;

END_SCHEMA_MAP;
```

In this example, view instances of type **ViewEntity** are created from every combination of base instances of type **BaseA**, **BaseB**, and **BaseC**, for which the **WHEN** clause is true. The **BEGIN_VIEW** clause defines the computations required to initialize the attributes of the new view entity instances that are created.

3.10 The FROM Clause

```
from_head = FROM '(' extended_entity_ref  { ',' extended_entity_ref }
            ')' .
```

The **FROM** clause defines the base entity types from which a new view instance is to be created and its attributes initialized. Note that a view instance can be created and its attributes initialized from many entity types, not just one, by listing multiple base entity types in the **FROM** clause.

The names of the entity types in the **FROM** clause are specified as extended entity references, where each reference identifies a variable name, schema name, and entity type name as defined above.

In the example mapping schema above (i.e., **Mapping_Schema**), the **FROM** clause has the form:

```
FROM (ba : bdb::BaseA, bb : bdb::BaseB, bc : bdb::BaseC)
```

All three entity types are defined in the base schema (i.e., **Base_Schema**) and can be referenced with the variable names **ba**, **bb**, and **bc**, respectively. In this example, the **FROM** clause means that the creation of view instances of type **ViewEntity** will be based on these three types of base entities in the base schema.

Conceptually, the **FROM** clause of a view definition defines an iteration over the instances of a set of base entity types. This iteration produces every combination of base instances for the base entity types listed in the **FROM** clause. For each combination of base instances, the **WHEN** clause is evaluated and, if true, a new view instance is created and its attributes initialized.

For example, the **FROM** clause above (i.e., **FROM (ba : bdb::BaseA, bb : bdb::BaseB, bc : bdb::BaseC)**) creates the following iteration during the materialization process:

```
for each {ba| ba is an instance of type bdb::BaseA}
      for each {bb| bb is an instance of type bdb::BaseB}
            for each {bc| bc is an instance of type bdb::BaseC}
                  begin
                        evaluate the WHEN clause for (ba, bb, bc)
                        if the WHEN clause is true
                              then create a new view instance of type
                                    ViewEntity and initialize its
                                    attributes
                  end
```

The scope of the variable in each extended entity reference in a **FROM** clause is the view declaration containing the **FROM** clause. The variable name must be unique for each extended entity reference in a **FROM** clause. The value of the variable is assigned as part of the iteration created by the **FROM** clause.

Note that a **VIEW** declaration creates a new instance of the view entity type for every combination of base instances listed in the **FROM** clause, unless the **WHEN** clause in the **VIEW** declaration evaluates to **FALSE** for a particular combination. There are no other restrictions imposed by a **VIEW** declaration on creation of new view instances. This means, for example, that if a **VIEW** declaration is executed twice, the same set of view instances is created twice. If this behavior is undesired, then it must be prevented using the **WHEN** clause for the **VIEW** declaration.

3.11 The WHEN Clause

```
when_clause = WHEN domain_rule ‘;’ { domain_rule ';' } .
```

The **WHEN** clause of a view declaration defines the conditions under which a new view instance is created and its attributes initialized. It consists of the keyword **WHEN** followed by one or more expressions, separated by semicolons. Each of these expressions is a domain rule as defined in EXPRESS.

Conceptually, the **WHEN** clause of a view declaration is evaluated for every combination of entity instances specified in the **FROM** clause (or **COMPOSE** clause, see below). For each combination that produces a value of **TRUE** for all the expressions in the **WHEN** clause, a new view entity is created and the values of its attributes initialized. The newly created view instance is assigned to the variable specified in the extended entity reference that defines the view entity type created by the view declaration containing the **WHEN** clause. This allows statements within the view declaration to refer to the new view instance (see below).

In the example mapping schema above (i.e., **Mapping_Schema**), the **WHEN** clause has the following form:

```
VIEW v : vdb::ViewEntity ;
FROM (ba : bdb::BaseA, bb : bdb::BaseB, bc : bdb::BaseC)
WHEN ((ba.int_a1 = bb.complex_b1.int_a1) AND
```

```
        (NOT (bc.real_c1 > 10.0)));
```

The expression in the parentheses is evaluated once for each combination of base instances of type **bdb::BaseA**, **bdb::BaseB**, and **bdb::BaseC** generated by the preceding **FROM** clause. For each combination of base instances for which the expression is **TRUE**, a new view instance of type **ViewEntity** is created. This new view instance is used to initialize variable **v** so that other parts of the view declaration can refer to the new view instance. Note that in the expression, **ba**, **bb**, and **bc** function as variables whose current value is defined by the current combination of base instances produced by the **FROM** clause iteration.

3.12 The Logical Organization of a Compose Mapping Declaration

Whereas the **VIEW** declaration defines an iteration over a set of base instances for the purpose of deriving new view instances from the base instances, the **COMPOSE** declaration is used to define an iteration over the view instances of a particular type that have already been created in a view model. This might be done, for example, to compute relationships between view instances that could not be computed earlier because not all the related instances had been created yet. Thus, the **COMPOSE** declaration is used to set up multiple passes in EXPRESS-X for computing the values of view instance attributes .

```
compose_decl = compose_head [algorithm_head] stmt {stmt}
               END_COMPOSE ';' .
```

The **COMPOSE** declaration begins with a header, and ends with the keyword **END_COMPOSE**. In-between is a series of statements computing the values of attributes for view instances of a particular type. As in a **VIEW** declaration, a **COMPOSE** declaration can contain local definitions that will be needed by the statements it contains.

```
compose_head = COMPOSE general_head [ from_head ] when_clause
               BEGIN_COMPOSE .

general_head = ( (name_id FOR extended_entity_ref) | extended_entity_ref)
               ';' .

name_id = simple_id .
```

The header for a **COMPOSE** declaration begins with the keyword **COMPOSE** and is followed by the name of a view entity type that has already been materialized. This view entity type name is specified as an extended entity reference (i.e., variable : schema::entity type). This creates an iteration over all view instances of that type. Like in a **VIEW** declaration, the view entity type named in the **COMPOSE** declaration can be preceded by a unique identifier and the keyword **FOR**. This provides a unique identification for the **COMPOSE** declaration when more than one such declaration is needed for the same view entity type.

Next in the **COMPOSE** declaration is an optional **FROM** clause. If present, this **FROM** clause augments the iteration described in the preceding paragraph by combining it with additional nested iterations for all combinations of instances of the entity types listed in the **FROM** clause. These iterations are similar to those created by the **FROM** clause in a **VIEW** declaration.

Next in the **COMPOSE** declaration header is a **WHEN** clause that defines the conditions that must be met by the current combination of entity instances in the iteration to apply the mapping defined in the rest of the **COMPOSE** declaration. This **WHEN** clause is defined exactly as discussed above.

Finally, the **COMPOSE** declaration header ends with the keyword **BEGIN_COMPOSE**.

Inside the **COMPOSE** declaration is a sequence of statements that defines how values for the attributes of a view instance should be computed. These statements are described in a later section.

As an example of a **COMPOSE** declaration, consider the following:

```
COMPOSE v : vdb::ViewEntity ;
WHEN (v.real_v1 > v.int_v1);
BEGIN_COMPOSE
   v.int_v1  := v.real_v2;
END_COMPOSE;
```

This **COMPOSE** declaration creates an iteration over view instances of type **ViewEntity**. During each iteration, the variable **v** is initialized with a different instance of this type.

3.13 Statements in View and Compose Mapping Declarations

The declarations **VIEW** and **COMPOSE** are somewhat analogous to the declaration of subprograms in programming languages. As such, they can contain constant declarations, local variable declarations and sequences of statements that specify the details of a mapping.

EXPRESS-X has sixteen types of statements for use inside **VIEW** and **COMPOSE** declarations. Many of these statement types are taken directly from EXPRESS. Those that are either not in EXPRESS or are modified from their definition in EXPRESS include: the assignment statement, the **FROM** statement, the **WHEN** statement, the initialize statement, the **DELETE** statement, and the instantiation statement.

```
stmt = alias_stmt | assignment_stmt | case_stmt | compound_stmt
       | delete_stmt | escape_stmt | from_stmt | if_stmt | init_stmt
       | instantiation_stmt | null_stmt | procedure_call_stmt
       | repeat_stmt | return_stmt | skip_stmt | when_stmt .
```

3.13.1 Enhanced Assignment Statement

```
assignment_stmt = [coercion] general_ref { qualifier } (':=' | '+=' | '-=')
                  expression ‘;’ .
```

An assignment statement is used to define the computation required to compute the value for an attribute in a mapping declaration and uses the typical assignment statement format found in programming languages such as Pascal and C. It is also similar to the assignment statement in EXPRESS. On the left of an assignment operator (e.g., ‘**:=**’) is the name of an entity attribute with any necessary qualifications (e.g., group or index qualifiers). On the right of the assignment operator is an expression, the value of which is used to initialize the attribute specified on the left of the assignment operator.

If the type of the value produced by the expression on the right of the assignment operator has a simple type (i.e., **Number**, **Integer**, **Real**, **Boolean**, **Logical**, **String**, or **Binary**), then this value is used to initialize the attribute specified to the left of the assignment operator. On the other hand, if the type of the value produced by the expression is not a simple type, then a pointer to the value is assigned to the attribute specified to the left of the assignment operator.

In addition to the standard assignment operator (i.e., ‘**:=**’), there are two special versions of it with the following meanings:

`A += expression`	is equivalent to	`A := A + expression`
`A -= expression`	is equivalent to	`A := A - expression`

These special forms of the assignment operator are useful for efficient memory management when assigning values to attributes that are aggregate types.

Examples of view assignment statements include:

```
real_v1 := 100.0 + BaseC\BaseA.int_a1 * 10.0;

int_v2  := {INTEGER} BaseC.real_c1;

ent_v3.str_attr := 'This is a view object';

agg_v4[4] := BaseC.agg_c1[0];

int_v5 += BaseA.int_b1;

ent_v6 := {BaseA} BaseB;

boolean_v7 := person IS man;
```

Note that examples 2 and 6 illustrate casting in expressions. This is explained in section 3.13.1.4.

3.13.1.1 Coercion in Assignment Statements

```
coercion = select_coercion | subtype_coercion .
```

Coercion may be used on the left-hand side of an assignment statement to specify a particular type that an attribute may take. Single braces are used to specify **SELECT** coercion and double braces are used for subtype coercion.

SELECT coercion is used to specify that an attribute that is a **SELECT** type should be a particular type from the selection:

```
select_coercion = '{' ( entity_id | type_id ) '}' .

entity_id = extended_id ;

type_id = extended_id ;
```

Example - if an entity geometric_item has an attribute geometry which may be a line, a bezier curve, or a b-spline curve:

```
TYPE curve = SELECT(b_spline, bezier, line);
END_TYPE;

ENTITY geometric_item;
         geometry : curve;
END_ENTITY;
```

The target instance for the assignment operation could be coerced into being a line:

```
VIEW v : vdb::geometric_item;
FROM (e : bdb::source_entity)
WHEN TRUE;
BEGIN_VIEW
         {line}geometry := e;
END_VIEW;
```

When an attribute refers to a supertype that may be instantiated as one of many subtypes, subtype coercion is used.

```
subtype_coercion = '{{' entity_id '}}' .
```

Example - the gender of a child.

```
ENTITY mother;
         name     : STRING;
         age      : INTEGER;
        ...
         youngest : child;
END_ENTITY;

ENTITY child ABSTRACT SUPERTYPE OF ( ONEOF (boy, girl));
         name : STRING;
         age : INTEGER;
END_ENTITY;

ENTITY boy SUBTYPE OF ( child );
       toy : STRING;
END_ENTITY;

ENTITY girl SUBTYPE OF ( child );
       doll : STRING;
END_ENTITY;
```

Then the target instance for the assignment operator could be coerced into either a boy or girl instance as appropriate:

```
VIEW m : vdb::mother;
FROM ( ... )
WHEN ... ;
BEGIN_VIEW
         ...
         IF new_born.sex = 'MALE' THEN
               {{ boy }} youngest .toy := 'Powerful Gun'
         ELSE
               {{ girl }} youngest .doll := 'Beautiful Princess'
         END_IF;
END_VIEW;
```

3.13.1.2 Enhancements to Expressions in Assignment Statements

EXPRESS-X extends EXPRESS expressions to introduce the `IS` keyword, casting, and references to manually instantiated entity instances.

```
expression = simple_expression [ rel_op_extended simple_expression ] .

rel_op_extended = rel_op | IN | LIKE | IS .

simple_expression = term { add_like_op term } .

add_like_op = '+' | '-' | OR | XOR .

term = factor { multiplication_like_op factor } .

multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .

factor = simple_factor [ '**' simple_factor ] .

simple_factor = aggregate_initializer | entity_constructor
                | enumeration_reference | interval | query_expression
                | ( [ unary_op ] ( '(' expression ')' | primary ) ) .

primary = literal | ( [cast] qualifiable_factor { qualifier } ) .

literal = binary_literal | integer_literal | logical_literal
          | real_literal | string_literal .

cast = '{' simple_types | entity_id | type_id '}' .

qualifiable_factor = attribute_ref | constant_factor | function_call
                     | general_ref | instance_ref | population .

instance_ref = instance_id .

instance_id = '#' extended_id .
```

3.13.1.3 The IS Operator

The **IS** operator in EXPRESS-X is used to determine if a particular instance of an attribute is of a particular type. It returns a boolean value.

```
rel_op_extended = rel_op | IN | LIKE | IS .
```

Example - is a particular instance of entity **person** also of entity type **man**?

```
boolean_v7 := person IS man;
```

3.13.1.4 Casting in Expressions

Attributes may be cast to a specified data type in an expressions using a cast operator. To do this, the attribute is preceded by the casting data type in braces. Defined types and entity types are cast using appropriate **VIEW** declarations and functions defined elsewhere in the mapping schema.

```
primary = literal | ( [cast] qualifiable_factor { qualifier } ) .

cast = '{' simple_types | entity_id | type_id '}' .
```

Rules and restrictions:

a) Casting to defined data types is only possible if a corresponding function is defined elsewhere in the mapping schema.

b) Casting to entity types is only possible if a corresponding **VIEW** declaration is defined elsewhere in the mapping schema, and the **FROM** clause for this view contains a single entity type which is the type to be cast (i.e., a conformance class 1 **VIEW** declaration). Alternately, a function can be defined that specifies the cast.

Example - the entity instance in the attribute **source_curve** is cast to a **bezier_curve** entity instance. A **VIEW** declaration must exist to carry out the conversion of the **source_curve** entity type to a **bezier_curve** entity type. This **VIEW** declaration must contain the **source_curve** entity type as the only entity type in its **FROM** clause, and it must be a **VIEW** declaration that creates instances of entity type **bezier_curve**.

```
target_curve := {bezier_curve} source_curve;
```

3.13.1.5 Reference to a Manually Instantiated Entity Instance

An expression can reference a previously created entity instance that was manually instantiated in the **GLOBAL** section of a mapping schema or using an instantiation statement (see below). This is done by placing a ‘#’ before the identifier of the manually instantiated instance.

```
qualifiable_factor = attribute_ref | constant_factor | function_call
                     | general_ref | instance_ref | population .

instance_ref = instance_id .

instance_id = '#' extended_id .
```

3.13.2 FROM Statements

```
from_stmt = from_head when_clause BEGIN stmt { stmt } END ';' .
```

The **FROM** statement defines an iteration process for computing the values for one or more attributes of a new view instance. It begins with a header that is identical in syntax with the **FROM** clause in the header of a **VIEW** declaration. In this case, however, the **FROM** clause identifies the entity instances to use to compute attribute values inside a **VIEW** or **COMPOSE** declaration. The **FROM** clause is followed by a **WHEN** clause, which is also identical in syntax with the **WHEN** clause in the header of a **VIEW** declaration.

Logically, the **FROM** statement creates an iteration for each entity type listed in its header. Each of these entity types is specified as an extended entity reference (i.e., variable : schema::entity type). The iterations are nested in the order that the extended entity references are specified from left to right. This has the effect of executing the **WHEN** clause for every combination of entity instances for the entity types listed in the **FROM** statement header. The variables in the extended entity references for these entity types are initialized appropriately for each iteration as discussed for the **FROM** clause above.

The **FROM** statement also logically defines a scope that is the scope of the variables in the extended entity references listed in the **FROM** statement header. Scopes are nested and a variable name is resolved using the inner most scope that contains the variable name (as is done in programming languages like Pascal and C).

An example of a **FROM** statement is shown below:

```
FROM (c : sdb::child, w : sdb::woman)
WHEN
   (c IN w.offspring);
BEGIN
   IF (c.sex = 'BOY')
      f.children += {tdb::boy}c;
   ELSE
      f.children += {tdb::girl}c;
   END_IF;
END;
```

This **FROM** statement creates an iteration over all combinations of entities of type **child** and **woman** from the **sdb** schema. For each **child** instance that is listed as an offspring of the **woman** instance, the instance is cast as an instance of entity type **boy** or **girl** in the **tdb** schema and added to the **children** aggregate defined in the encompassing scope, which also defines the variable **f**.

3.13.3 WHEN Statements

```
when_stmt = when_clause BEGIN stmt {stmt} END ';' .
```

A **WHEN** statement in the body of a **VIEW** or **COMPOSE** declaration is similar to an **IF** statement. It defines the conditions under which other statements should be executed. This is useful, for example, when the value to be assigned to an attribute of a view instance is computed differently in various cases.

The **WHEN** statement begins with a **WHEN** clause as defined above for a **VIEW** declaration header. This clause is followed by a **BEGIN** block containing any number of statements. These statements are executed when the **WHEN** clause evaluates to **TRUE**.

As an example of using the **WHEN** statement in the body of a **VIEW** declaration, consider the following example mapping schema:

```
SCHEMA_MAP Mapping_Schema;

GLOBAL

   DECLARE bdb INSTANCE OF Base_Schema;
   DECLARE vdb INSTANCE OF View_schema;

END_GLOBAL;

VIEW v : vdb::ViewEntity
FROM (b : bdb::BaseB)
WHEN (b.str_b2 = 'SUPPLIER');
BEGIN_VIEW

    WHEN
```

```
        (EXISTS(b.complex_b1)
        AND
        (b.complex_b1.int_a1 > 100)
        );
    BEGIN
        v.int_v1 := b.complex_b1.int_a1;
    END;

    v.str_v2 := BaseB.str_b2;

END_VIEW;

END_SCHEMA_MAP;
```

In this example, new view instances of type **ViewEntity** are created in the view model from base instances of type **BaseB** in the base model. For each of these new view instances, if the **complex_b1** attribute of the base instance from which it is created exists and has an attribute **int_a1** with a value greater than 100, then this value is used to initialize the **int_v1** attribute of the new view instance. If either of these conditions is false, then the **int_v1** attribute of the new view instance is not initialized. The **str_v2** attribute of new view instances is always initialized.

3.13.4 Initialize Statement

```
init_stmt = NEW general_ref { qualifier } ';' .
```

Conceptually, the initialize statement is similar to a constructor in the C++ programming language. It creates a persistent instance of a non-primitive data type. The statement begins with the keyword **NEW**. This keyword is followed by the data type to be created, expressed as a variable name with any necessary qualifications (e.g., attribute, group or index qualifiers). It is often useful for initializing aggregates so that items can be added to them in a mapping.

Note that the **FROM** clause in a **VIEW** declaration header has an implicit initialize statement in it to create a new view instance and assign it to the variable that is part of the extended entity reference that specifies the view entity type for the **VIEW** declaration.

An example of the initialize statement is the following:

```
    NEW f.children;
```

This creates a new empty instance of the **children** aggregate entity type used above.

3.13.5 DELETE Statement

```
delete_stmt = delete_instance_stmt .
```

Conceptually, the **DELETE** statement is similar to a destructor in object-oriented programming languages. It deletes a persistent instance of a non-primitive data type. The statement begins with the keyword **DELETE**.

```
delete_instance_stmt = DELETE general_ref {qualifier} ‘;’ .
```

When deleting an entity instance, the **DELETE** keyword is followed by the entity instance to be deleted, expressed as a variable name with any necessary qualifications (e.g., attribute, group or index qualifiers).

An example of the **DELETE** statement is the following:

```
DELETE female;
```

If **female** is a variable inside the iteration of a **FROM** statement, for example, then the statement deletes the instance that is currently bound to this variable.

3.13.6 Instantiation Statement

```
instantiation_stmt = instantiation_clause .
```

Entity instances can be manually instantiated in the **GLOBAL** section of a mapping schema. They can also be manually instantiated inside **VIEW** and **COMPOSE** declarations using an instantiation statement. The syntax is identical to the syntax used to manually instantiate entity instances in the **GLOBAL** section of a mapping schema.

3.14 The Logical Organization of a Member Mapping Declaration

A **MEMBER** declaration is the final new type of declaration that can appear in a mapping schema. It is used to identify and logically group a set of attributes from a view entity type. This group helps to establish a relationship between a base schema and a view schema. Such a logical group of attributes typically has some practical meaning for the application systems that will use the view once materialized.

For example, a **MEMBER** declaration could be used to define a group of all the entity instances belonging to single assembly within a larger product model. The entity instances in this logical group can then be treated as an atomic unit when appropriate (e.g., for check-in and check-out functions). As another example, an information processing system may use the group of attributes contained in a **MEMBER** declaration to decide when the values of a view model should be recomputed in response to changes in the corresponding base model. A third example of the use of a **MEMBER** declaration is to specify which entity instances should be copied from a base model to a view model to make a deep copy during the view materialization process.

The group of attributes in a **MEMBER** declaration is given a name and each attribute in the group is given a label. If one of the attributes in a group references another entity type, then this attribute represents the root of a tree of entity types. The **MEMBER** declaration has a clause that prunes the branches of this tree.

```
member_decl = member_head BEGIN_MEMBER [ include_clause ]
              [ exclude_clause ] END_MEMBER ';' .

member_head = MEMBER general_head [ from_head ] [ when_clause ] .

general_head = ( (name_id FOR extended_entity_ref) | extended_entity_ref)
               ';' .

name_id = simple_id .
```

A **MEMBER** declaration begins with the keyword **MEMBER** followed by the name of an entity type specified as an extended entity reference. This entity type contains the attributes that are the roots for the trees of entities that make up the member group. Optionally, the extended entity reference can be preceded by the keyword **FOR** and a name to uniquely identify the **MEMBER** declaration.

Optionally, the **MEMBER** declaration includes a **FROM** clause and/or a **WHEN** clause. As for the **COMPOSE** declaration, if a **FROM** clause is specified, it adds additional nested iterations to the top level iteration for the entity type containing the root attributes of the member group. The **WHEN** clause, if present, identifies the conditions that must be satisfied by a particular combination of entity instances in order to apply the body of the **MEMBER** declaration to it.

The specific attributes to be included in the member group are identified by the **INCLUDE** clause. The clause lists each attribute to include in the group, gives it a unique label, and specifies the type of the attribute.

As mentioned above, if one of the included attributes references another view entity type, then this attribute is the root of a tree of entity types. The branches of this tree are pruned with the **EXCLUDE** clause. The clause specifies the path through the tree to the attribute of an entity that is to be pruned. Each of the paths to an attribute to prune is given a label, and the type of the attribute to be pruned is specified.

```
include_clause = INCLUDE member_component { member_component } .

exclude_clause = EXCLUDE member_component { member_component } .

member_component = member_attr_stmt | member_when_stmt;

member_when_stmt = when_clause BEGIN member_component {member_component}
                   END ';' .

member_attr_stmt = label ':'  parameter_type ':=' (SELF | attribute_ref)
                   { qualifier } ';' .
```

Optionally, a **WHEN** clause can be used with **INCLUDE** and **EXCLUDE** to identify the conditions that must be true for the **INCLUDE** or **EXCLUDE** to be applied.

As an example consider the following **MEMBER** declaration:

```
MEMBER assembly_position_mem FOR arm_component_assembly_position ;
BEGIN_MEMBER
INCLUDE
   attr1 : cap_item
              := off;
   attr2 : context_dependent_shape_representation
              := context_dependent_shape_representation_ptr;
EXCLUDE
   WHEN
         ((SELF.off IS shape_representation_relationship)
          OR
         ((SELF.off IS representation_relationship_with_transpormation)) ;
   BEGIN
   attr101 : representation
                  := off\representation_relationship.rep_1;
   attr102 : representation
                  := off\representation_relationship.rep_2;
   attr103 : product_definition_shape
                  := context_dependent_shape_representation_ptr.
                           represented_product_relation;
   END;
```

```
   WHEN
         (SELF.off IS mapped_item);
   BEGIN
   attr104 : representation
                  := off.mapping_source.mapped_representation;
   END;

END_MEMBER;
```

This **MEMBER** declaration defines a group of attributes from the view entity type **arm_component_assembly_position**. The name of this group is **assembly_position_mem**. This group includes two attributes from the **arm_component_assembly_position** entity type: **off** and **context_dependent_shape_representation_ptr**. The group labels for these attributes are **attr1** and **attr2**, respectively. The **EXCLUDE** clause identifies cases where attributes of the entity types referenced by these two attributes are to be pruned from the group.

3.15 Structure of a Mapping Schema

A typical structure for a mapping schema is to have one or more **VIEW** mapping declarations (representing pass one of the materialization process) in which all view instances are created, followed by zero or more **COMPOSE** mapping declarations that compute values for the uninitialized attributes in these view instances. At least one **COMPOSE** mapping declaration is needed for each view entity type with uninitialized attributes at the end of pass one. After all **COMPOSE** mapping declarations, all view instances in a view model are required to be valid. Finally **MEMBER** declarations, if needed, are placed at the end of a mapping schema. This is illustrated below:

```
SCHEMA_MAP Mapping_Schema;

GLOBAL

   DECLARE sdb INSTANCE OF source_Schema;
   DECLARE tdb INSTANCE OF target_schema;

END_GLOBAL;

(* Beginning of Pass 1 - Create view instances *)

VIEW v1 : tdb::ViewEntity1 ;
. . .
END_VIEW;

VIEW v2 : tdb::ViewEntity2 ;
. . .
END_VIEW;

. . .

VIEW vn : tdb::ViewEntityn ;
. . .
END_VIEW;


(* Beginning of Pass 2 and later passes - Initialize the *)
```

```
(* uninitialized attributes in the new view instances *)

COMPOSE v1 : tdb::ViewEntity1 ;
. . .
END_COMPOSE;

COMPOSE v2 : tdb::ViewEntity2 ;
. . .
END_COMPOSE;

. . .

COMPOSE vn : tdb::ViewEntityn ;
. . .
END_COMPOSE;


(* Beginning of definition of attribute groups for *)
(* entity types in the view schema *)

MEMBER membership1 FOR vi : tdb::ViewEntityi ;
. . .
END_MEMBER;

MEMBER membership2 FOR vi : tdb::ViewEntityi ;
. . .
END_MEMBER;

. . .

END_SCHEMA_MAP;
```

Appendix A: EXPRESS-X Example 1

Base Schema

```
(*
This schema defines the structure of the data stored in the base model.
i.e. entity names, attribute names and types.

The key features are:
Entity person has an attribute data that is either a man or woman entity.
Entity woman has a list of children.
The gender of each child entity is given by the attribute sex which may
    take the value BOY or GIRL.
*)


SCHEMA source;

TYPE m_or_f = SELECT (man, woman);
END_TYPE;

TYPE b_or_g = ENUMERATION OF (BOY, GIRL);
END_TYPE;

ENTITY person;
        social_security_number : STRING (8) fixed
        name : STRING;
        age : REAL;
        data : m_or_f;
END_ENTITY;

ENTITY man;
        car : STRING;
        pocket_contents : wallet;
END_ENTITY;

ENTITY woman;
        offspring : LIST [0:?] OF child;
        handbag_contents : wallet;
END_ENTITY;

ENTITY wallet;
        credit_card  : STRING;
        num_twenties : INTEGER;
        num_tens     : INTEGER;
        num_fives    : INTEGER;
        total_change : REAL;
END_ENTITY;
```

```
ENTITY child;
        name : STRING;
        age  : REAL;
        sex  : b_or_g;
END_ENTITY;

END_SCHEMA;
```

View Schema

```
(*
This schema defines the structure of data in the view model.

The key features are:
A female entity has a list of dependants, which is an abstract supertype of
    either a boy or girl entity.
*)

SCHEMA target;

ENTITY male;
         id      : STRING;
         age     : INTEGER;
         vehicle : STRING;
         wallet  : money_bag;
END_ENTITY;

ENTITY female;
         id       : STRING;
         age      : INTEGER;
         children : LIST [0:?] OF dependant;
         purse    : money_bag;
END_ENTITY;

ENTITY dependant ABSTRACT SUPERTYPE OF ( ONEOF (boy, girl) );
         age : INTEGER;
         name : STRING;
END_ENTITY;

ENTITY money_bag;
         plastic : STRING;
         total_cash : REAL;
END_ENTITY;

ENTITY boy SUBTYPE OF ( dependant );
END_ENTITY;

ENTITY girl SUBTYPE OF ( dependant );
END_ENTITY;

END_SCHEMA;
```

Mapping Schema

```
(*** Mapping_Schema ***)
SCHEMA_MAP Mapping_Schema;

GLOBAL
  DECLARE sdb INSTANCE OF source; (*** instance of base schema ***)
  DECLARE tdb INSTANCE OF target; (*** instance of view schema ***)
END_GLOBAL;

(*** male view scope ***)

VIEW l : tdb::male ;
FROM (m : sdb::man)
WHEN TRUE;
BEGIN_VIEW

  FROM (p : sdb::person)
  WHEN
     ((p.data IS sdb::man)
      AND
      (p.data = m));
  BEGIN
        id := p.social_security_number;
  END;

  vehicle := m.car;

  NEW l.wallet;
  wallet.plastic    := m.pocket_contents.credit_card;
  wallet.total_cash := m.pocket_contents.num_twenties * 20.0 +
                       m.pocket_contents.num_tens * 10.0 +
                       m.pocket_contents.num_fives * 5.0 +
                       m.pocket_contents.total_change;

END_VIEW;

END_SCHEMA_MAP;
```

Appendix B: EXPRESS-X Example 2

Base Schema

Same as in Example 1 in Appendix A.

View Schema

Same as in Example 1 in Appendix A.

Mapping Schema

```
SCHEMA_MAP Mapping_Schema2;

GLOBAL
  DECLARE sdb INSTANCE OF source; (*** instance of base schema ***)
  DECLARE tdb INSTANCE OF target; (*** instance of view schema ***)

  #tdb::extra_child = tdb::boy(2, 'Tony Blurb');
END_GLOBAL;

VIEW b : tdb::boy ;
FROM ( c : sdb::child )
WHEN
  ( c.sex = ‘BOY’ ) ;
BEGIN_VIEW
  age := {INTEGER} child.age;
  name := child.name;
END_VIEW;

VIEW g : tdb::girl ;
FROM ( c : sdb::child )
WHEN
  ( c.sex = ‘GIRL’ ) ;
BEGIN_VIEW
  age := {INTEGER} child.age;
  name := child.name;
END_VIEW;


VIEW f : tdb::female ;
FROM (w : sdb::woman)
WHEN TRUE;
BEGIN_VIEW

  FROM (p : sdb::person)
  WHEN
```

```
     ((p.data IS sdb::woman)
      AND
      (p.data = w));
  BEGIN
         id := p.social_security_number;
  END;

  NEW f.children;

  FROM (c : sdb::child)
  WHEN
    (c IN w.offspring) ;
  BEGIN
    IF (c.sex = 'BOY') THEN
       children += {tdb::boy}child;
    ELSE
       children += {tdb::girl}child;
    END_IF;
  END;

  children += #extra_child;

  NEW f.purse;
  purse.plastic := w.handbag_contents.credit_card;
  purse.total_cash := w.handbag_contents.num_twenties * 20.0 +
                      w.handbag_contents.num_tens * 10.0 +
                      w.handbag_contents.num_fives * 5.0 +
                      w.handbag_contents.total_change;


END_VIEW;


END_SCHEMA_MAP;
```

Appendix C: EXPRESS-X Example 3

Base Schema

```
(*
EXPRESS schema defining Base Model
*)

SCHEMA source_schema;

ENTITY family;
         family_name: STRING;
         members: LIST [1:?] OF person;
END_ENTITY;

ENTITY person
   ABSTRACT SUPERTYPE OF (ONEOF(man, woman, child));
        name : STRING;
        age : INTEGER;
END_ENTITY;

ENTITY man
   SUBTYPE OF (person);
        car : STRING;
        pocket_contents : wallet;
END_ENTITY;

ENTITY woman
   SUBTYPE OF (person);
        handbag_contents : wallet;
END_ENTITY;

ENTITY wallet;
        credit_card  : STRING;
        num_twenties : INTEGER;
        num_tens     : INTEGER;
        num_fives    : INTEGER;
        total_change : REAL;
END_ENTITY;

ENTITY child
   SUBTYPE OF (person);
END_ENTITY;

END_SCHEMA;
```

View Schema

```
(*
EXPRESS schema defining View Model
*)

SCHEMA target_schema;

ENTITY family_member
   ABSTRACT SUPERTYPE OF (ONEOF(husband, wife, dependant));
         family_id : STRING;
         name        : STRING;
         age         : INTEGER;
END_ENTITY;

ENTITY husband
   SUBTYPE OF (family_member);
         wife_is     : wife;
         children : LIST[0:?] OF dependant;
         vehicle  : STRING;
         wallet   : money_bag;
END_ENTITY;

ENTITY wife
   SUBTYPE OF (family_member);
         husband_is   : husband;
         children     : LIST[0:?] OF dependant;
         purse        : money_bag;
END_ENTITY;

ENTITY dependant
   SUBTYPE OF (family_member);
         father_is : husband;
         mother_is: wife;
         siblings : LIST[0:?] OF dependant;
END_ENTITY;

ENTITY money_bag;
         plastic : STRING;
         total_cash : REAL;
END_ENTITY;


END_SCHEMA;
```

Mapping Example

```
SCHEMA_MAP mapping_schema;

GLOBAL
  DECLARE sdb INSTANCE OF source_schema; (*** instance of base schema ***)
  DECLARE tdb INSTANCE OF target_target; (*** instance of view schema ***)
END_GLOBAL;

VIEW h : tdb::husband ;
FROM (f : sdb::family, m : sdb::man)
WHEN
  (m IN f.members);
BEGIN_VIEW

  family_id := 'Family_of_' + f.family_name;
  name := m.name;
  age  := m.age;
  vehicle := m.car;

  NEW h.wallet;
  wallet.plastic    := m.pocket_contents.credit_card;
  wallet.total_cash := m.pocket_contents.num_twenties * 20.0 +
                       m.pocket_contents.num_tens * 10.0 +
                       m.pocket_contents.num_fives * 5.0 +
                       m.pocket_contents.total_change;

END_VIEW;


VIEW wife : tdb::wife ;
FROM (f : sdb::family, w : sdb::woman)
WHEN
   (w IN f.members);
BEGIN_VIEW

  family_id := 'Family_of_' + f.family_name;
  name := w.name;
  age  := w.age;

  NEW wife.purse;
  purse.plastic := w.handbag_contents.credit_card;
  purse.total_cash := w.handbag_contents.num_twenties * 20.0 +
                      w.handbag_contents.num_tens * 10.0 +
                      w.handbag_contents.num_fives * 5.0 +
                      w.handbag_contents.total_change;

END_VIEW;


VIEW d : tdb::dependant ;
FROM (f : sdb::family, c : sdb::child)
WHEN
   (c IN f.members);
```

```
BEGIN_VIEW

  family_id := 'Family_of_' + f.family_name;
  name := c.name;
  age := c.age;

END_VIEW;


COMPOSE h : tdb::husband ;
WHEN TRUE;
BEGIN_COMPOSE

  FROM (w : tdb::wife)
  WHEN
     (h.family_id = w.family_id);
  BEGIN

    wife_is := w;

  END;

  NEW h.children;
  FROM (d : tdb::dependant)
  WHEN
     (h.family_id = d.family_id);
  BEGIN

    children += d;

  END;

END_COMPOSE;

COMPOSE w : tdb::wife ;
WHEN TRUE;
BEGIN_COMPOSE

  FROM (h : tdb::husband)
  WHEN
     (w.family_id = h.family_id);
  BEGIN

    husband_is := h;

  END;

  NEW w.children;
  FROM (d : tdb::dependant)
  WHEN
     (w.family_id = d.family_id);
  BEGIN

    children += d;
```

```
  END;

END_COMPOSE;


COMPOSE d : tdb::dependant ;
WHEN TRUE;
BEGIN_COMPOSE

  FROM (h : tdb::husband)
  WHEN
     (h.family_id = d.family_id);
  BEGIN

    father_is := h;

  END;

  FROM (w : tdb::wife)
  WHEN
     (w.family_id = d.family_id);
  BEGIN

    mother_is := w;

  END;

  NEW d.siblings;
  FROM (d1 : tdb::dependant)
  WHEN
     (d1.family_id = d.family_id)
     AND
     NOT (d1 = d);
  BEGIN

    siblings += d1;

  END;

END_COMPOSE;



END_SCHEMA_MAP;
```

Appendix D: EXPRESS-X Example 4

Base Schema

```
SCHEMA config_control_design; (* AP203 AIM Schema *)

         ...

END_SCHEMA;
```

View Schema

```
SCHEMA ap203_arm_schema; (* AP203 ARM Schema *)

         ...

ENTITY arm_part;

(* POINTERS INTO THE AIM *)
  off : product;

  product_category_relationship_ptr : product_category_relationship;
  product_related_product_category_ptr : product_related_product_category;

(* USER DEFINED ATTRIBUTES *)
  arm_key : STRING;
  arm_user_name : STRING;
  arm_product_description : STRING;
  arm_part_nomenclature : STRING;
  arm_part_number : STRING;
  arm_standard_part_indicator : STRING;
  arm_part_type : STRING;

(* RELATIONSHIPS TO OTHER ARM OBJECTS *)
  arm_to_alternate_part : LIST [0:?] OF arm_part;
  arm_is_alternate_part_for : LIST [0:?] OF arm_part;
  arm_to_part_version : LIST [1:?] OF arm_part_version;

(* POINTERS FROM OTHER arm OBJECTS *)
  arm_to_person : arm_person;
  arm_to_application_context : LIST [0:?] OF arm_application_context;

UNIQUE
  UR1 : arm_key;

END_ENTITY;

         ...
END_SCHEMA;
```

Mapping Schema

```
SCHEMA_MAP AP203_aim2arm_mapping_schema;(* AP203 AIM To ARM Mapping *)

GLOBAL
   DECLARE aim_db INSTANCE OF config_control_design;
   DECLARE arm_db INSTANCE OF ap203_arm_schema;
END_GLOBAL;


VIEW np : arm_db::arm_part ;
FROM (p : aim_db::product)
WHEN TRUE;
BEGIN_VIEW

   off := p;
   arm_product_description := p.description;

   FROM (prpc : aim_db::product_related_product_category)
   WHEN (p IN prpc.products);
   BEGIN
      arm_part_type := NVL(prpc.name, '') + ' - ' +
                          NVL(prpc.description, '');
      product_related_product_category_ptr := prpc;
      WHEN (prpc\product_category.name = 'standard_part');
      BEGIN
         arm_standard_part_indicator
           := NVL(prpc\product_category.name, '') + ' - ' +
                NVL(prpc\product_category.description, '');
      END;
   END;

   FROM (prpc : aim_db::product_related_product_category,
         pcr : aim_db::product_category_relationship)
   WHEN
         (p IN prpc.products)
         AND
         (prpc\product_category
          = pcr.sub_category);
   BEGIN
         product_category_relationship_ptr := pcr;
   END;

   arm_part_nomenclature := p.name;

   arm_part_number       := p.id;

END_VIEW;


COMPOSE np : arm_db::arm_part ;
WHEN TRUE;
BEGIN_COMPOSE
```

```
   NEW np.arm_to_application_context;
   FROM (nac : arm_db::arm_application_context,
         pc : aim_db::product_context)
   WHEN
         (pc IN np.off.frame_of_reference)
         AND
         (pc\application_context_element.frame_of_reference
               = nac.off);
   BEGIN
         arm_to_application_context += nac;
   END;

   NEW np.arm_to_alternate_part;
   FROM (nap : arm_db::arm_alternate_part)
   WHEN
         (nap.off.base
               = np.off);
   BEGIN
      arm_to_alternate_part += np;
   END;

   NEW np.arm_is_alternate_part_for;
   FROM (nap : arm_db::arm_alternate_part)
   WHEN
         (nap.off.alternate
               = np.off);
   BEGIN
      arm_is_alternate_part_for += np;
   END;

   NEW np.arm_to_part_version;
   FROM (npv : arm_db::arm_part_version)
   WHEN (npv.off\product_definition_formation.of_product = np.off);
   BEGIN
      arm_to_part_version += arm_part_version;
   END;

   FROM (arm_person : arm_db::arm_person)
   WHEN
         EXISTS(arm_person.arm_to_person_item)
         AND
         (np IN arm_person.arm_to_person_item);
   BEGIN
         arm_to_person := arm_person;
   END;

  arm_key := NVL(np.arm_part_number, 'NO VALUE GIVEN');

  arm_user_name := NVL(np.off.name, 'NO VALUE GIVEN');

END_COMPOSE;
```

```
MEMBER part_membership FOR p : arm_db::arm_part ;
BEGIN_MEMBER
INCLUDE
   attr1 : product
         := off;
   attr2 : product_related_product_category
         := product_related_product_category_ptr;
   attr3 : product_category_relationship
         := product_category_relationship_ptr;
EXCLUDE
   attr101 : SET [1:?] OF product_context
               := off.frame_of_reference;
   attr102 : SET [1:?] OF product
               := product_related_product_category_ptr.products;
   attr103 : product_category
               := product_category_relationship_ptr.category;
END_MEMBER;

END_SCHEMA_MAP; (* End Of ap203_aim2arm_mapping_schema *)
```

Appendix E: EXPRESS-X Example 5

Base Schema

```
Same as in Example 1 in Appendix A.
```

View Schema

```
Same as in Example 1 in Appendix A.
```

Mapping Schema

```
Same as in Example 1 in Appendix A.
```

Updating Schema

```
(* This example shows how to propagate the updates in view model back
   to the source model
*)

(*** Updating_Schema ***)
SCHEMA_MAP Updating_Schema;

GLOBAL
  DECLARE sdb INSTANCE OF source; (*** instance of base schema ***)
  DECLARE tdb INSTANCE OF target; (*** instance of view schema ***)
END_GLOBAL;

(*** male update scope ***)

COMPOSE tm : tdb::male ;
WHEN TRUE;
BEGIN_COMPOSE

  FROM (p : sdb::person, m : sdb::man)
  WHEN
     ((p.social_security_number = tm.id)
      AND
      (p.data IS sdb::man)
      AND
      (p.data = m));
  BEGIN
     p.social_security_number := tm.id;
     m.car := tm.vehicle;
```

```
    m.pocket_contents.credit_card := tm.wallet.plastic;

    (* Notice that the following values can not be uniquely decided.
      The following only shows one possible solution.
    *)

    m.pocket_contents.num_twenties := tm.wallet.total_cash / 20;
    m.pocket_contents.num_tens
      := (tm.wallet.total_cash-20 * m.pocket_contents.num_twenties) / 10;
    m.pocket_contents.num_fives
      := (tm.wallet.total_cash - 20 * m.pocket_contents.num_twenties
         - 10 * m.pocket_contents.num_tens) / 5 ;
    m.pocket_contents.total_change
      := tm.wallet.total_cash - 20 * m.pocket_contents.num_twenties
         - 10 * m.pocket_contents.num_tens
         - 5  * m.pocket_contents.num_fives ;

  END;

END_COMPOSE;

END_SCHEMA_MAP;
```

Appendix F: EXPRESS Language Syntax

F.1 Tokens

F.1.1 Keywords

```
 0 | ABS = 'abs' .
 1 | ABSTRACT = 'abstract' .
 2 | ACOS = 'acos' .
 3 | AGGREGATE = 'aggregate' .
 4 | ALIAS = 'alias' .
 5 | AND = 'and' .
 6 | ANDOR = 'andor' .
 7 | ARRAY = 'array' .
 8 | AS = 'as' .
 9 | ASIN = 'asin' .

10 | ATAN = 'atan' .
11 | BAG = 'bag' .
12 | BEGIN = 'begin' .
13 | BINARY = 'binary' .
14 | BLENGTH = 'blength' .
15 | BOOLEAN = 'boolean' .
16 | BY = 'by' .
17 | CASE = 'case' .
18 | CONSTANT = 'constant' .
19 | CONST_E = 'const_e' .

20 | CONTEXT = 'context' .
21 | COS = 'cos' .
22 | DERIVE = 'derive' .
23 | DIV = 'div' .
24 | ELSE = 'else' .
25 | END = 'end' .
26 | END_ALIAS = 'end_alias' .
27 | END_CASE = 'end_case' .
28 | END_CONSTANT = 'end_constant' .
29 | END_CONTEXT = 'end_context' .

30 | END_ENTITY = 'end_entity' .
31 | END_FUNCTION = 'end_function' .
32 | END_IF = 'end_if' .
33 | END_LOCAL = 'end_local' .
34 | END_MODEL = 'end_model' .
35 | END_PROCEDURE = 'end_procedure' .
36 | END_REPEAT = 'end_repeat' .
37 | END_RULE = 'end_rule' .
38 | END_SCHEMA = 'end_schema' .
39 | END_TYPE = 'end_type' .

40 | ENTITY = 'entity' .
41 | ENUMERATION = 'enumeration' .
```

```
42 | ESCAPE = 'escape' .
43 | EXISTS = 'exists' .
44 | EXP = 'exp' .
45 | FALSE = 'false' .
46 | FIXED = 'fixed' .
47 | FOR = 'for' .
48 | FORMAT = 'format' .
49 | FROM = 'from' .

50 | FUNCTION = 'function' .
51 | GENERIC = 'generic' .
52 | HIBOUND = 'hibound' .
53 | HIINDEX = 'hiindex' .
54 | IF = 'if' .
55 | IN = 'in' .
56 | INSERT = 'insert' .
57 | INTEGER = 'integer' .
58 | INVERSE = 'inverse' .
59 | LENGTH = 'length' .

60 | LIKE = 'like' .
61 | LIST = 'list' .
62 | LOBOUND = 'lobound' .
63 | LOINDEX = 'loindex' .
64 | LOCAL = 'local' .
65 | LOG = 'log' .
66 | LOG10 = 'log10' .
67 | LOG2 = 'log2' .
68 | LOGICAL = 'logical' .
69 | MOD = 'mod' .

70 | MODEL = 'model' .
71 | NOT = 'not' .
72 | NUMBER = 'number' .
73 | NVL = 'nvl' .
74 | ODD = 'odd' .
75 | OF = 'of' .
76 | ONEOF = 'oneof' .
77 | OPTIONAL = 'optional' .
78 | OR = 'or' .
79 | OTHERWISE = 'otherwise' .

80 | PI = 'pi' .
81 | PROCEDURE = 'procedure' .
82 | QUERY = 'query' .
83 | REAL = 'real' .
84 | REFERENCE = 'reference' .
85 | REMOVE = 'remove' .
86 | REPEAT = 'repeat' .
87 | RETURN = 'return' .
88 | ROLESOF = 'rolesof' .
89 | RULE = 'rule' .

90 | SCHEMA = 'schema' .
```

```
 91 | SELECT = 'select' .
 92 | SELF = 'self' .
 93 | SET = 'set' .
 94 | SIN = 'sin' .
 95 | SIZEOF = 'sizeof' .
 96 | SKIP = 'skip' .
 97 | SQRT = 'sqrt' .
 98 | STRING = 'string' .
 99 | SUBTYPE = 'subtype' .

100 | SUPERTYPE = 'supertype' .
101 | TAN = 'tan' .
102 | THEN = 'then' .
103 | TO = 'to' .
104 | TRUE = 'true' .
105 | TYPE = 'type' .
106 | TYPEOF = 'typeof' .
107 | UNIQUE = 'unique' .
108 | UNKNOWN = 'unknown' .
109 | UNTIL = 'until' .

110 | USE = 'use' .
111 | USEDIN = 'usedin' .
112 | VALUE = 'value' .
113 | VALUE_IN = 'value_in' .
114 | VALUE_UNIQUE = 'value_unique' .
115 | VAR = 'var' .
116 | WHERE = 'where' .
117 | WHILE = 'while' .
118 | XOR = 'xor' .
```

F.1.2 Character classes

```
119 | bit = '0' | '1' .

120 | digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
121 | digits = digit { digit } .
122 | encoded_character = octet octet octet octet.
123 | hex_digit = digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' .
124 | letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' |
             'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' |
             'u' | 'v' | 'w' | 'x' | 'y' | 'z' .
125 | lparen_not_star = '(' not_star .
126 | not_lparen_star = not_paren_star | ')' .
127 | not_paren_star = letter | digit | not_paren_star_special .
128 | not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' |
                                   '+' | ',' | '-' | '.' | '/' | ':' |
                                   ';' | '<' | '=' | '>' | '?' | '@' |
                                   '[' | '\' | ']' | '^' | '_' | '`' |
                                   '{' | '|' | '}' | '~' .
129 | not_paren_star_special = not_paren_star_quote_special | '''' .
```

```
130 | not_quote = not_paren_star_quote_special | letter | digit | '(' |
                ')' | '*' .
131 | not_rparen = not_paren_star | '*' | '(' .
132 | not_star = not_paren_star | '(' | ')' .
133 | octet = hex_digit hex_digit .
134 | special = not_paren_star_quote_special | '(' | ')' | '*' | '''' .
135 | star_not_rparen = '*' not_rparen .
```

F.1.3 Lexical Elements

```
136 | binary_literal = '%' bit { bit } .
137 | encoded_string_literal = '"' encoded_character { encoded_character }
                              '"' .
138 | integer_literal = digits .
139 | real_literal = digits '.' [ digits ] [ 'e' [ sign ] digits ] .

140 | simple_id = letter { letter | digit | '_' } .
141 | simple_string_literal = \q { ( \q \q ) | not_quote | \s | \o } \q .
```

F.1.4 Remarks

```
142 | embedded_remark = '(*' { not_lparen_star | lparen_not_star |
                        star_not_rparen | embedded_remark } '*)' .
143 | remark = embedded_remark | tail_remark .
144 | tail_remark = '--' { \a | \s | \o } \n .
```

F.1.5 Interpreted Identifiers

```
145 | attribute_ref = attribute_id .
146 | constant_ref = constant_id .
147 | entity_ref = entity_id .
148 | enumeration_ref = enumeration_id .
149 | function_ref = function_id .

150 | parameter_ref = parameter_id .
151 | procedure_ref = procedure_id .
152 | schema_ref = schema_id .
153 | type_label_ref = type_label_id .
154 | type_ref = type_id .
155 | variable_ref = variable_id .
```

F.2 Grammar Rules

```
156 | abstract_supertype_declaration = ABSTRACT SUPERTYPE
                                       [subtype_constraint] .
157 | actual_parameter_list = '(' parameter { ',' parameter } ')' .
```

```
158 | add_like_op = '+' | '-' | OR | XOR .
159 | aggregate_initializer = '[' [ element { ',' element }] ']' .

160 | aggregate_source = simple_expression .
161 | aggregate_type = AGGREGATE [ ':' type_label ] OF parameter_type .
162 | aggregation_types = array_type | bag_type | list_type | set_type .
163 | algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
164 | alias_stmt = ALIAS variable_id FOR general_ref { qualifier } ';'
                 stmt { stmt } END_ALIAS ';' .
165 | array_type = ARRAY '[' bound_spec ']' OF [ OPTIONAL ] [ UNIQUE ]
                 base_type .
166 | assignment_stmt = general_ref { qualifier } ':=' expression ';' .
167 | attribute_decl = attribute_id | qualified_attribute .
168 | attribute_id = simple_id .
169 | attribute_qualifier = '.' attribute_ref .

170 | bag_type = BAG [ bound_spec ] OF base_type .
171 | base_type = aggregation_types | simple_types | named_types .
172 | binary_type = BINARY [ width_spec ] .
173 | boolean_type = BOOLEAN .
174 | bound_1 = numeric_expression .
175 | bound_2 = numeric_expression .
176 | bound_spec = '[' bound_1 ':' bound_2 ']' .
177 | built_in_constant = CONST_E | PI | SELF | '?' .
178 | built_in_function = ABS | ACOS | ASIN | ATAN | BLENGTH | COS  |
                        EXISTS | EXP | FORMAT | HIBOUND | HIINDEX |
                        LENGTH | LOBOUND | LOINDEX | LOG | LOG2   |
                        LOG10 | NVL | ODD | ROLESOF | SIN |SIZEOF |
                        SQRT | TAN | TYPEOF | USEDIN | VALUE | VALUE_IN|
                        VALUE_UNIQUE .
179 | built_in_procedure = INSERT | REMOVE .

180 | case_action = case_label { ',' case_label } ':' stmt .
181 | case_label = expression .
182 | case_stmt = CASE selector OF { case_action } [ OTHERWISE ':' stmt ]
                END_CASE ';' .
183 | compound_stmt = BEGIN stmt { stmt } END ';' .
184 | constant_body = constant_id ':' base_type ':=' expression ';' .
185 | constant_decl = CONSTANT constant_body { constant_body }
                    END_CONSTANT ';' .
186 | constant_factor = built_in_constant | constant_ref .
187 | constant_id = simple_id .
188 | constructed_types = enumeration_type | select_type .
189 | declaration = entity_decl | function_decl | procedure_decl |
                  type_decl .

190 | derived_attr = attribute_decl ':' base_type ':=' expression ';' .
191 | derive_clause = DERIVE derived_attr { derived_attr } .
192 | domain_rule = [ label ':' ] expression.
193 | element = expression [ ':' repetition ] .
194 | entity_body = { explicit_attr } [ derive_clause ] [ inverse_clause ]
                  [ unique_clause ] [ where_clause ] .
195 | entity_constructor = entity_ref '(' [ expression { ',' expression} ]
```

```
                         ')' .
196 | entity_decl = entity_head entity_body END_ENTITY;
197 | entity_head = ENTITY entity_id [ subsuper ] ';' .
198 | entity_id = simple_id .
199 | enumeration_id = simple_id .

200 | enumeration_reference = [ type_ref '.' ] enumeration_ref .
201 | enumeration_type = ENUMERATION OF '(' enumeration_id
                       { ',' enumeration_id } ')' .
202 | escape_stmt = ESCAPE ';' .
203 | explicit_attr = attribute_decl { ',' attribute_decl } ':' [OPTIONAL]
                    base_type ';' .
204 | expression = simple_expression [rel_op_extended simple_expression] .
205 | factor = simple_factor [ '**' simple_factor ] .
206 | formal_parameter = parameter_id { ',' parameter_id } ':'
                       parameter_type .
207 | function_call = ( built_in_function | function_ref )
                    [ actual_parameter_list ] .
208 | function_decl = function_head [algorithm_head] stmt { stmt }
                    END_FUNCTION ';' .
209 | function_head = FUNCTION function_id [ '(' formal_parameter
                    { ';' formal_parameter } ')' ] ':' parameter_type
                    ';' .

210 | function_id = simple_id .
211 | generalized_types = aggregate_type | general_aggregation_types |
                        generic_type .
212 | general_aggregation_types = general_array_type | general_bag_type |
                                general_list_type | general_set_type .
213 | general_array_type = ARRAY [ bound_spec] OF [ OPTIONAL ] [ UNIQUE ]
                         parameter_type .
214 | general_bag_type = BAG [ bound_spec ] OF parameter_type .
215 | general_list_type = LIST [ bound_spec ] OF [ UNIQUE ]
                        parameter_type.
216 | general_ref = parameter_ref | variable_ref .
217 | general_set_type = SET [ bound_spec ] OF parameter_type .
218 | generic_type = GENERIC [ ':' type_label ] .
219 | group_qualifier = '\' entity_ref .

220 | if_stmt = IF expression THEN stmt { stmt } [ ELSE stmt { stmt } ]
              END_IF ';' .
221 | increment = numeric_expression .
222 | increment_control = variable_id ':=' bound_1 TO bound_2 [ BY
                        increment ] .
223 | index = numeric_expression .
224 | index_1 = index .
225 | index_2 = index .
226 | index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
227 | integer_type = INTEGER .
228 | interface_specification = reference_clause | use_clause .
229 | interval = '{' interval_low interval_op interval_item interval_op
               interval_high '}' .

230 | interval_high = simple_expression .
```

```
231 | interval_item = simple_expression .
232 | interval_low = simple_expression .
233 | interval_op = '<' | '<=' .
234 | inverse_attr = attribute_decl ':' [ ( SET | BAG ) [bound_spec] OF ]
                     entity_ref FOR attribute_ref ';' .
235 | inverse_clause = INVERSE inverse_attr { inverse_attr } .
236 | label = simple_id .
237 | list_type = LIST [ bound_spec ] OF [ UNIQUE ] base_type .
238 | literal = binary_literal | integer_literal | logical_literal |
                real_literal | string_literal .
239 | local_decl = LOCAL local_variable { local_variable } END_LOCAL ';' .

240 | local_variable = variable_id { ',' variable_id } ':' parameter_type
                       [ ':=' expression ] ';' .
241 | logical_expression = expression .
242 | logical_literal = FALSE | TRUE | UNKNOWN .
243 | logical_type = LOGICAL .
244 | multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .
245 | named_types = entity_ref | type_ref .
246 | named_type_or_rename = named_types [ AS ( entity_id | type_id ) ] .
247 | null_stmt = ';' .
248 | number_type = NUMBER .
249 | numeric_expression = simple_expression .

250 | one_of = ONEOF '(' supertype_expression { ',' supertype_expression }
               ')' .
251 | parameter = expression .
252 | parameter_id = simple_id .
253 | parameter_type = generalized_types | named_types | simple_types .
254 | population = entity_ref .
255 | precision_spec = numeric_expression .
256 | primary = literal | ( qualifiable_factor { qualifier } ) .
257 | procedure_call_stmt = ( built_in_procedure | procedure_ref )
                            [ actual_parameter_list ] ';' .
258 | procedure_decl = procedure_head [ algorithm_head ] { stmt }
                       END_PROCEDURE ';' .
259 | procedure_head = PROCEDURE procedure_id [ '(' [VAR] formal_parameter
                       { ';' [ VAR ] formal_parameter } ')' ] ';' .

260 | procedure_id = simple_id .
261 | qualifiable_factor = attribute_ref | constant_factor | function_call
                           | general_ref | population.
262 | qualified_attribute = SELF group_qualifier attribute_qualifier .
263 | qualifier = attribute_qualifier | group_qualifier
                  | index_qualifier .
264 | query_expression = QUERY '(' variable_id '<*' aggregate_source '|'
                         logical_expression ')' .
265 | real_type = REAL [ '(' precision_spec ')' ] .
266 | referenced_attribute = attribute_ref | qualified_attribute .
267 | reference_clause = REFERENCE FROM schema_ref ['(' resource_or_rename
                         { ',' resource_or_rename } ')' ] ';' .
268 | rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .
269 | rel_op_extended = rel_op | IN | LIKE .
```

```
270 | rename_id = constant_id | entity_id | function_id | procedure_id |
                type_id .
271 | repeat_control = [ increment_control ] [ while_control ]
                     [ until_control ] .
272 | repeat_stmt = REPEAT repeat_control ';' stmt { stmt } END_REPEAT
                  ';' .
273 | repetition = numeric_expression .
274 | resource_or_rename = resource_ref [ AS rename_id ] .
275 | resource_ref = constant_ref | entity_ref | function_ref |
                   procedure_ref | type_ref .
276 | return_stmt = RETURN [ '(' expression ')' ] ';' .
277 | rule_decl = rule_head [ algorithm_head ] { stmt } where_clause
                END_RULE ';' .
278 | rule_head = RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')'
                ';' .
279 | rule_id = simple_id .

280 | schema_body = { interface_specification } [ constant_decl ]
                  { declaration | rule_decl } .
281 | schema_decl = SCHEMA schema_id ';' schema_body END_SCHEMA ';' .
282 | schema_id = simple_id .
283 | selector = expression .
284 | select_type = SELECT '(' named_types { ',' named_types } ')' .
285 | set_type = SET [ bound_spec ] OF base_type .
286 | sign = '+' | '-' .
287 | simple_expression = term { add_like_op term } .
288 | simple_factor = aggregate_initializer | entity_constructor |
                    enumeration_reference | interval |
                    query_expression | ( [ unary_op ] ( '(' expression
                    ')' | primary ) ) .
289 | simple_types = binary_type | boolean_type | integer_type |
                   logical_type | number_type | real_type |
                   string_type .

290 | skip_stmt = SKIP ';' .
291 | stmt = alias_stmt | assignment_stmt | case_stmt | compound_stmt |
           escape_stmt | if_stmt | null_stmt | procedure_call_stmt |
           repeat_stmt | return_stmt | skip_stmt .
292 | string_literal = simple_string_literal | encoded_string_literal .
293 | string_type = STRING [ width_spec ] .
294 | subsuper = [ supertype_constraint ] [ subtype_declaration ] .
295 | subtype_constraint = OF '(' supertype_expression ')' .
296 | subtype_declaration = SUBTYPE OF '(' entity_ref { ',' entity_ref }
                          ')' .
297 | supertype_constraint = abstract_supertype_declaration |
                           supertype_rule .
298 | supertype_expression = supertype_factor { ANDOR supertype_factor } .
299 | supertype_factor = supertype_term { AND supertype_term } .

300 | supertype_rule = SUPERTYPE subtype_constraint .
301 | supertype_term = entity_ref | one_of | '(' supertype_expression
                     ')' .
302 | syntax = schema_decl { schema_decl } .
303 | term = factor { multiplication_like_op factor } .
```

```
304 | type_decl = TYPE type_id '=' underlying_type ';' [ where_clause ]
                END_TYPE ';' .
305 | type_id = simple_id .
306 | type_label = type_label_id | type_label_ref .
307 | type_label_id = simple_id .
308 | unary_op = '+' | '-' | NOT .
309 | underlying_type = constructed_types | aggregation_types |
                      simple_types | type_ref .

310 | unique_clause = UNIQUE unique_rule ';' ( unique_rule ';' } .
311 | unique_rule = [ label ':' ] referenced_attribute { ','
                  referenced_attribute } .
312 | until_control = UNTIL expression .
313 | use_clause = USE FROM schema_ref [ '(' named_type_or_rename
                 { ',' named_type_or_rename } ')' ] ';' .
314 | variable_id = simple_id .
315 | where_clause = WHERE domain_rule ';' { domain_rule ';' } .
316 | while_control = WHILE logical_expression .
317 | width = numeric_expression .
318 | width_spec = '(' width ')' [ FIXED ] .
```

Appendix G: EXPRESS-X Extensions to the EXPRESS Language

G.1 Tokens Added

```
BEGIN_COMPOSE = 'begin_compose' .
BEGIN_MEMBER = 'begin_member' .
BEGIN_VIEW = 'begin_view' .
COMPOSE = 'compose' .
DECLARE = 'declare' .
DELETE = 'delete' .
END_COMPOSE = 'end_compose' .
END_GLOBAL = 'end_global' .
END_MEMBER = 'end_member' .
END_SCHEMA_MAP = 'end_schema_map' .
END_VIEW = 'end_view' .
EXCLUDE = 'exclude' .
GLOBAL = 'global' .
INSTANCE = 'instance' .
IS = 'is' .
MEMBER = 'member' .
NEW = 'new' .
SCHEMA_MAP = 'schema_map' .
VIEW = 'view' .
WHEN = 'when' .
```

G.2 Syntax Rules Added

```
cast = '{' simple_types | entity_id | type_id '}' .

coercion = select_coercion | subtype_coercion .

compose_decl = compose_head [ algorithm_head ] stmt {stmt}
               END_COMPOSE ';' .

compose_head = COMPOSE general_head [ from_head ] when_clause
               BEGIN_COMPOSE .

delete_instance_stmt = DELETE general_ref { qualifier } ‘;’ .

delete_stmt = delete_instance_stmt .

exclude_clause = EXCLUDE member_component { member_component } .

extended_entity_ref  =  variable_id ':' parameter_type .

extended_id = [schema_id '::'] simple_id .

from_head = FROM '(' extended_entity_ref { ',' extended_entity_ref }  ')' .
```

```
from_stmt = from_head when_clause BEGIN stmt { stmt } END ';' .

general_head = ( (name_id FOR extended_entity_ref) | extended_entity_ref )
               ';' .

global_decl = GLOBAL { schema_instance_decl | instantiation_clause }
              END_GLOBAL ';' .

include_clause = INCLUDE member_component { member_component } .

init_stmt = NEW general_ref { qualifier } ';' .

instance_id = '#' extended_id .

instance_ref = instance_id

instantiation_clause = instance_id '=' entity_constructor ';' .

instantiation_stmt = instantiation_clause;

member_attr_stmt = [ label ':' ] parameter_type ':=' (SELF | attribute_ref)
                   { qualifier } ';' .

member_component = member_attr_stmt | member_when_stmt;

member_decl = member_head BEGIN_MEMBER [ include_clause ]
              [ exclude_clause ] END_MEMBER ';' .

member_head = MEMBER general_head [ from_head ] [ when_clause ] .

member_when_stmt = when_clause BEGIN member_component { member_component }
                   END ';' .

name_id = simple_id .

schema_instance_decl = DECLARE schema_instance_id INSTANCE OF schema_id
               ';' .

schema_instance_id = simple_id .

select_coercion = '{' ( entity_id | type_id ) '}' .

subtype_coercion = '{{' entity_id '}}' .

view_decl = view_head [ algorithm_head ] { stmt } END_VIEW ';' .

view_head = VIEW general_head from_head when_clause BEGIN_VIEW .

when_clause = WHEN domain_rule ';' { domain_rule ';' } .

when_stmt = when_clause BEGIN stmt { stmt } END ';' .
```

G.3 Modifications or Extensions To The Existing EXPRESS Syntax Rules

```
assignment_stmt = [coercion] general_ref { qualifier } (':=' | '+=' | '-=')
                  expression ‘;’ .

declaration = compose_decl | entity_decl | function_decl | member_decl
              | procedure_decl | type_decl | view_decl .

entity_id = extended_id .

primary = literal | ( [cast] qualifiable_factor { qualifier } ) .

qualifiable_factor = attribute_ref | constant_factor | function_call |
                     general_ref | instance_ref | population .

rel_op_extended = rel_op | IN | LIKE | IS .

schema_body = { interface_specification } [constant_decl] { global_decl }
              { declaration | rule_decl } .

schema_decl = SCHEMA_MAP schema_id ‘;’ schema_body END_SCHEMA_MAP ‘;’ .

stmt = alias_stmt | assignment_stmt | case_stmt | compound_stmt |
       delete_stmt | escape_stmt | from_stmt | if_stmt | init_stmt |
       instantiation_stmt | null_stmt | procedure_call_stmt | repeat_stmt |
       return_stmt | skip_stmt | when_stmt .

type_id = extended_id .
```